# Toward Self-Adjusting $k$-ary Search Tree Networks

**Evgeniy Feder**
ITMO University, Russia

**Anton Paramonov**
EPFL, Switzerland

**Pavel Mavrin**
Paphos University, Cyprus

**Iosif Salem**
TU Berlin, Germany
ZeroPoint Technologies AB

**Vitaly Aksenov**
City, University of London, UK

**Stefan Schmid**
TU Berlin, Germany
Fraunhofer SIT, Germany

## Abstract

Datacenter networks are becoming increasingly flexible with the incorporation of new optical communication technologies, such as optical circuit switches, enabling self-adjusting topologies that can adapt to the traffic pattern in a demand-aware manner. In this paper, we take the first steps toward demand-aware and self-adjusting $k$-ary tree networks. These are more powerful generalizations of existing binary search tree networks (like SplayNet [22]), which have been at the core of self-adjusting network (SAN) designs. $k$-ary search tree networks are a natural generalization offering nodes of higher degrees, reduced route lengths, and local routing in spite of reconfigurations (due to maintaining the search property).

Our main results are two online heuristics for self-adjusting $k$-ary tree networks. Empirical results show that our heuristics work better than SplayNet in most of the real network traces and for average to low locality synthetic traces, and are only a little inferior to SplayNet in all remaining traces. We build our online algorithms by first solving the offline case. First, we compute an offline (optimal) static demand-aware network for arbitrary traffic patterns in $\mathcal{O}(n^3 \cdot k)$ time via dynamic programming, where $n$ is the number of network nodes (e.g., datacenter racks), and also improve the bound for the special case of uniformly distributed traffic. Then, we present a centroid-based approach to demand-aware network designs that we use both in the offline static and online settings. In the offline uniform-workload case, we construct this centroid network in linear time $\mathcal{O}(n)$.

**2012 ACM Subject Classification** Networks → Network algorithms; Networks → Network design principles; Theory of computation → Online algorithms; Theory of computation → Data structures design and analysis

**Keywords and phrases** self-adjusting networks, networks, splay-tree, k-ary tree

**Digital Object Identifier** 10.4230/LIPIcs.ESA.2024.21

**Funding** This research was partially supported by JetBrains and by German Federal Ministry of Education and Research (BMBF), grant 16KISK020K (6G-RIC), 2021-2025.

32nd Annual European Symposium on Algorithms (ESA 2024).
Editors: Timothy Chan, Johannes Fischer, John Iacono, and Grzegorz Herman; Article No. 21; pp. 21:1–21:15
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

# 1 Introduction

With more services being offloaded to the cloud and the ever increasing numbers of devices connected to the Internet, inter- and intra-datacenter traffic is growing explosively. Therefore, datacenter network design has been attracting a lot of attention. Traditional datacenter network designs are static and perform well only under certain workloads (e.g. all-to-all). However, datacenter traffic follows patterns which can be exploited in the design of more efficient networks. It has been shown that a small fraction of network nodes accounts for a large fraction of datacenter traffic, the traffic distribution is sparse, and it exhibits locality features that change over time [2].

As a result, innovative dynamic datacenter network topologies have emerged. Supported by advances in networking hardware (e.g., optical circuit switches, 60 GHz wireless, or even experimental designs based on free-space optics as in [15]), physical network topologies now have the ability to *self-adjust*. That is, the physical network topology is now programmable and can be reconfigured to serve traffic more efficiently. Leading cloud providers have already started incorporating dynamic networks into their datacenters [15, 20].

This flexibility provided by networking hardware raises an optimization challenge: how to optimally adjust the topology to improve network routing efficiency? There is a trade-off between the cost of changing the network topology (reconfiguration cost) and the benefit of reducing the distance (routing cost) of frequently communicating racks, henceforth called nodes. We assume that the input of this optimization problem is a sequence of communication requests. In the online case, where the input is revealed piecewise and future communication demand is unknown, we would opt for topology updates that are likely to pay off in the future. In the static case, we aim at computing an optimal demand-aware network topology with low time complexity.

The developing field of Self-Adjusting Networks (SANs) aims to address these optimization challenges. SANs often assume a family of allowed topologies, e.g., trees [1, 14, 10, 19, 22], skip lists [7], bounded degree graphs [5, 9], etc., within which the network has to remain. This restriction is not only practically motivated (e.g., optical switches are of bounded degree), but also simplifies algorithm design and allows for theoretical performance guarantees. Specifically, self-adjusting tree networks have been at the core of SAN designs. SplayNet [22], a self-adjusting binary search tree network generalizing splay trees [24], was the first proposed SAN. SplayNet has been extended to ReNet [9], a statically optimal SAN for sparse communication patterns, but also to a distributed version, DiSplayNet [19]. The search property is particularly useful for SANs, since maintaining it allows local and greedy routing despite changes in the topology.

Online SAN algorithms can vary from being fully reactive, in which case they reconfigure the topology after every communication request [7, 22] (e.g., when traffic is bursty), to being partially reactive [14], in which case they update the topology periodically. In [14], the topology changes every time the routing cost reaches a threshold $\alpha$ since the last topology update, the new topology is computed using SplayNet, and it remains static until the routing cost reaches the threshold again. This approach can be generalized to a meta-algorithm, where the topology changes upon a new chunk of the input, a subroutine is used to decide the new topology, and in between reconfigurations it remains static. Therefore, the efficient computation of static *demand-aware* topologies is also relevant in online SAN algorithm design.

In this work, we take the first steps to generalizing binary to $k$-ary search tree networks, since they provide higher node degrees and shorter routes than binary search trees (BSTs) for a fixed number of nodes, in addition to local and greedy routing regardless of reconfigurations due to the search property. We present offline static and online self-adjusting networks and evaluate our newly proposed SANs experimentally.

We note that designing $k$-ary search tree networks is different than designing a $k$-ary search tree. This holds due to the need of having a node identifier that stays the same across rotations in the network case (the assignment of identifiers to nodes is a bijection and does not change) as each tree node represents a network node. That is, in the case of $k$-ary search trees each node contains up to $k-1$ keys which are used for routing (traversing from root to searched key) in the data structure and are also the data. In contrast, in the network case each node can use up to $k-1$ nodes' keys for routing (hence called routing keys), but each node should have an extra fixed key that serves as its identifier (node key) which is the data. Self-adjusting $k$-ary search trees have been studied, for example, by Sherk in [23] and by Martel in [18]. However, none of the existing approaches, to the best of our knowledge, apply in our case due to the requirement of having one key (identifier) per node.

**Contributions.** We present offline static and online self-adjusting $k$-ary search tree networks. As for the static results, we have:

1. We construct a static "almost-optimal" $k$-ary tree network in $O(n^3 \cdot k)$ time using dynamic programming. Then, we reduce the complexity to $O(n^2 \cdot k)$ for the special case of uniformly distributed traffic to obtain optimal tree. The latter case is non-trivial, since we do not restrict the topology to full balanced trees.

2. We present a new static $k$-ary tree network topology which is built by $k+1$ trees with almost equal size connected around a centroid node. We present a linear offline algorithm that constructs it and prove that its total cost is close to the optimal.

As for the dynamic case, we have:

1. We then present two novel variants for online $k$-ary self-adjusting search tree networks: $k$-ary SplayNet, which is a generalization of SplayNet, and $(k+1)$-SplayNet which is obtained by applying a centroid heuristic. For both networks, we propose novel rotation operations that allow network nodes to keep their identifiers across tree rotations. To achieve the latter, we distinguish the set of node identifiers from the set of routing keys. We show how the known upper bounds for SplayNet can be applied for our networks.

2. We perform two types of experiments with synthetic network traces and real ones from datacenter network traffic. First, we show that our $k$-ary SplayNet indeed has better routing cost than SplayNet. Then, we compare 3-SplayNet to SplayNet, and to static demand-aware and demand-oblivious trees. The results show that 3-SplayNet (centroid topology) performs better than the standard SplayNet in most real network traces and in synthetic ones that have average to low temporal locality, while its performance is similar to SplayNet for the remaining datasets.

**Related work.** Self-Adjusting Networks were introduced with SplayNet [22]. SplayNet is a binary search tree network that generalizes Splay Trees [24]. SplayNet uses the tree rotations of splay trees to reduce the distance of communicating nodes to one; it uses splay operations to move the source and destination to their lowest common ancestor for each communication request. The same paper presents a dynamic programming algorithm for computing an optimal binary search tree network when the demand is known, among other results. SANs were further surveyed and classified in [8]. The authors present a classification of network topologies, which depends on whether they are (i) oblivious to or aware of traffic patterns, (ii) fixed or reconfigurable, and (iii) aware of the input sequence of communication requests (offline, online, generated by a distribution). This taxonomy allows for optimizing for certain properties according to each case, e.g. diameter or competitive ratio. A survey including first solutions and enabling networking technologies can be found in [17].

Tree-based SANs were further studied following SplayNet, due to being more easy to analyze and deploy. ReNets [9] are bounded-degree SANs based on combining ego-trees, which are trees where the source is a node and the remaining nodes are the destinations to which the source node has communicated with. In this design, ego-trees are stars or splay trees, depending on whether the number of destinations exceeds the degree bound. ReNets achieve static optimality for sparse communication patterns, which is a desirable optimality property [3, 8]. Ego-trees were further studied in the form of self-adjusting single-source tree networks in [1, 6], which provided a number of constant competitive (dynamically optimal) randomized and deterministic algorithms with good experimental performance. SplayNet has also been the basis of distributed tree SANs [10, 19] and of SANs in a cost model with non-unit cost for changing a link in the topology [14]. All the results mentioned above are for binary tree networks.

Alternative directions have also been studied. [4] studied how to construct offline SANs when the demand is known, under certain assumptions on the communication patterns (e.g. sparse demand). [16] presents a topology adjustment algorithm that uses static or dynamic topologies according to the identified traffic patterns (latency-sensitive, all-to-all, elephant flows). SANs that are not tree-based have also been studied, e.g. Skip List Networks [7].

**Paper organization.** In Section 2, we introduce all the necessary definitions. In Section 3, we explain how to build demand-aware optimal tree network using dynamic programming and how to build a quasi-optimal tree for the uniform workload. In Section 4, we present novel rotations for $k$-ary SplayNet and present two heuristics. In Section 5, we experimentally evaluate the cost of our new network structures. Finally, we conclude in Section 6.

## 2 Model

We consider a network of $n$ nodes $V = \{1, \ldots, n\}$ (e.g., top-of-the-rack switches in a datacenter networks) and a finite or infinite communication sequence $\sigma = (\sigma_1, \sigma_2, \ldots)$, where $\sigma_t = (u, v) \in V^2$ is a communication request from source $u$ to destination $v$. The network topology $G$ must be chosen from a family of desired topologies $\mathcal{G}$, for example, search trees, expander graphs, etc. Each topology $G \in \mathcal{G}$ is a graph $G = (V, E)$. The routing cost of $\sigma_t$ is given by the distance between the two endpoints in the topology when serving the request. The topology can be reconfigured between requests with a cost equal to the number of links (edges) added or removed. The total service cost of $\sigma$ is the sum of routing and reconfiguration costs. Our goal is to serve the communication sequence with minimum total cost.

We distinguish two problem variants. In the **offline static** variant, $\sigma$ is known in the form of an $n \times n$ demand matrix $D$, but no reconfiguration can occur. The matrix entry $D[u, v]$ is the number of requests from $u$ to $v$ in $\sigma$. We have to build a network topology $G_{static} \in \mathcal{G}$ that does not change during or in between requests. Such a graph $G_{static}$ needs to optimize the total distance function,

$$\text{TotalDistance}(D, G_{static}) = \sum_{(u,v)\in[n]^2} d_{G_{static}}(u, v) \cdot D[u, v]$$

where $d_{G_{static}}(u, v)$ is the distance between nodes $u$, $v$ in $G_{static}$ and $[n] = \{1, \ldots, n\}$.

In the **online self-adjusting** variant, $\sigma$ is not known in advance but revealed piecewise, and we can change the topology after serving a request. We are provided with an arbitrary initial network (before the first request arrives), which we denote by $G_0 \in \mathcal{G}$. Our task is to build an online algorithm $\mathcal{A}$ that adjusts the network $G_i$ at every time instant $i = 1, \dots, m$ and minimizes the total cost, which is calculated as $\text{sumCost}(\mathcal{A}, G_0, \sigma) = \sum_{i=1}^{m} (\text{routingCost}(G_{i-1}, \sigma_i) + \text{adjustmentCost}(G_{i-1},\ G_i))$, where $\text{routingCost}(G_{i-1}, \sigma_i)$ is the path length in edges of $G_{i-1}$ to route request $\sigma_i$ and $\text{adjustmentCost}(G_{i-1}, G_i)$ is the adjustment cost to reconfigure the network from step $i-1$, $G_{i-1} \in \mathcal{G}$, to step $i$, $G_i \in \mathcal{G}$, i.e., the number of edges added or deleted. Note that $\mathcal{A}$ can skip a reconfiguration step.

This paper focuses on both problem variants when the set of allowed topologies $\mathcal{G}$ is the set of $k$-ary search trees. These trees are the generalization of binary search trees, which were investigated in [22] in the context of SANs. The main advantage of using search trees as self-adjusting network topologies is that we can route locally and greedily: given a destination identifier (or address), each node can decide locally to which neighbor to forward the packet using the search property. This is particularly useful in the online setting, as routing tables do not need to be updated upon reconfiguration: a node given a packet can just use the information from the routing keys to forward the packet, accordingly. Also, with increasing $k$, route lengths decrease and node degrees increase.

▶ **Definition 1.** *(i) A* $k$-ary Search Tree *is a rooted tree on keys (node identifiers)* $1, \dots, n$*, where each node stores a key (node identifier), a routing array* $r = (r_1, r_2, \dots, r_{k-1})$ *containing routing elements (not keys), and has at most* $k$ *children defined by* $r$ *as follows: keys of nodes in the* $i$*-th child are between* $r_i$ *and* $r_{i+1}$ *for* $i \in [1, k-2]$*, keys of nodes to the left of* $r_1$ *are smaller than* $r_1$*, and keys of nodes to the right of* $r_{k-1}$ *are larger than* $r_{k-1}$*. Note that the key does not necessarily belong in the routing array.*

*(ii) A* routing-based $k$*-ary Search Tree is a* $k$*-ary search tree in which the node identifiers are contained in the routing array.*

Definition 1 defines the standard search route in a $k$-ary tree, when starting from the root. A routing path between two nodes is the unique path connecting them. Routing from a node containing source routing key $i$ to a destination node containing routing key $j$ occurs by following the upward (reverse search) path until their lowest common ancestor and then the standard downward search path to the destination. The local transformations of search tree networks are called rotations. A rotation in a $k$-ary search tree changes some adjacency relationships, while keeping subtrees intact and maintaining the search property (Definition 1). Such rotations can be implemented in a different manner. We introduce $k$-splay, a novel rotation procedure, in Section 4.

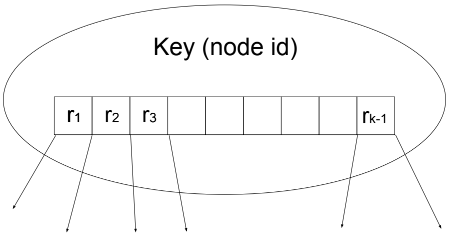

**Figure 1** Node in $k$-ary search tree

## 3 Optimal static $k$-ary search tree networks

In this section we construct an optimal static *routing-based* $k$-ary search tree network via dynamic programming. By analysing the optimal tree for a uniform workload, we show a linear-time but non-exact offline static algorithm that constructs a topology with a centroid node and $k+1$ trees connected to it for the chosen workload. The latter result will be the basis for an online self-adjusting network proposed in Section 4.

## 3.1 Dynamic programming algorithms

As our first result, we construct an offline (optimal) static routing-based $k$-ary search tree network. The algorithm is similar to the one for binary search tree from [22]: it just works as the dynamic programming on segments but a little bit more involved. Its complexity is $O(n^3 \cdot k)$.

The idea is that we compute the dynamic programming $dp[i][j][t]$ — what is the minimum cost if we partition the elements from $i$ to $j$ into $t$ disjoint trees. For $2 \leq t \leq k$, we can calculate it by adding trees by one: $dp[i][j][t] = \min\limits_{l \in [i, j-1]} (dp[i][l][1] + dp[l+1][j][t-1])$. When $t = 1$ the calculation is a little bit more complicated. We need to choose a root with the value $r$ on the interval $[i, j]$, split the left part of the interval $[i, r-1]$ on $d_l$ trees, and split the right part of the interval $[r+1, j]$ on $d_r$ trees with $d_l + d_r \leq k$, and, finally, we need to account for the total number of requests coming through the root $W[i, j]$, i.e., the number of requests outside the interval. The last part can be precalculated in $O(n^3)$ time as shown in [22]. As the formula, we get: $dp[i][j][1] = \min\limits_{r \in [i,j]} \min\limits_{d_l + d_r \leq k} (dp[i][r-1][d_l] + dp[r+1][j][d_r]$.

▶ **Theorem 2.** *An offline static routing-based $k$-ary search tree network, i.e., one with the minimal total distance given the requests in advance, can be constructed in $O(n^3 \cdot k)$.*

▶ **Remark 3.** *We remark that computing an optimal static non-routing-based $k$-ary search tree network is an open problem. That is, there is neither a trivial dynamic programming extension for it nor an NP-hardness proof.*

Then, we consider a uniform workload. That workload is an infinite workload where each pair of nodes is requested uniformly at random. In this case, we can improve the algorithm to $O(n^2 \cdot k)$ since the dynamic programming in this case does not depend on the position of the segment and depends only on its length, i.e., we can replace $dp[i][j]$ with $dp[j - i + 1]$. Our goal is to find a static $k$-ary search tree that serves an infinite uniform workload as fast as possible, i.e., the expectation of the cost of each query is minimal. So, we can represent the uniform workload as a workload with the demand matrix filled with ones.

▶ **Theorem 4.** *An offline static $k$-ary search tree network for the uniform workload can be constructed in $O(n^2 \cdot k)$.*

We remark that the resulting tree for the uniform workload is not required to be *routing-based* as in the generic case.

More details on these algorithms appear in the full version of the paper [12].

## 3.2 Centroid static $k$-ary search tree network in $O(n)$ time on uniform workload

Now, we present a linear-time construction of an almost-optimal offline $k$-ary search tree network for the uniform workload. We show the cost difference to the optimal tree in Theorem 6 and comment on experimental results in Remark 10. This construction will serve us as a basis of an online heuristic in Section 4. In general, faster computations of static network topologies are relevant for scaling to larger self-adjusting networks for which we compute new (demand-aware) topologies periodically. Also, the uniform workload is relevant to the all-to-all traffic pattern.

A $k$-ary search tree can be split in levels: the $i$-th level consists of nodes that are at distance $i - 1$ from the root. The tree is *weakly-complete* when all its levels, except for the last one, are fully filled (i.e., the $i$-th level has $k^{i-1}$ nodes). Nodes on the last level can be distributed arbitrarily as long as the search property holds.

Since in this case we consider only the uniform workload we can ignore the search property: we can first fix the tree structure and then distribute the keys so that the search property is respected. Our goal is to find an optimal $(k+1)$-degree tree (instead of a $k$-ary search tree) that minimizes $\text{TotalDistance}(D_{uniform}, T) = \sum_{(u,v)\in[n]\times[n]} d_T(u,v)$. A $(k+1)$-degree tree is a non-rooted tree where each node has at most $k+1$ neighbours. Such trees represent the same set of trees as $k$-ary search trees: you can root a $(k+1)$-degree tree by a leaf and obtain a $k$-ary search tree (by correctly distributing keys). Intuitively, we propose a topology where the root has $k+1$ children, in contrast to common a $k$-ary tree with a $k$-degree root, with the potential of reducing the total cost of routing communication requests.

▶ **Definition 5.** *A* centroid $(k+1)$*-degree tree is a tree with the root having* $k+1$ *weakly-complete* $k$*-ary trees. All the levels of the tree are fully filled, possibly except for the last one. We can change the relative positions of subtrees such that the leaves on the last level are all grouped together to the left. The tree is shown on Figure 2.*

We prove (Theorem 6) that the centroid $(k+1)$-degree tree $T$ has total distance in the uniform workload close to the total distance in the optimal $(k+1)$-degree tree. Intuitively, the proof works as follows. First, we show that all sibling subtrees, i.e., subtrees with the same parent, should either have the same height or their heights should be different by one. If not, we can move the leaf from one subtree to another and decrease the total cost. The latter move does not always decrease the cost, but sometimes it leads to a small overhead. Then, we show that if for some subtree not all leaves on the last level are aligned to the "left", then we can move them and decrease the cost. By that we show that the centroid tree is almost optimal. The complete proof is provided in the full version of the paper [12]. Here we just state the main results.

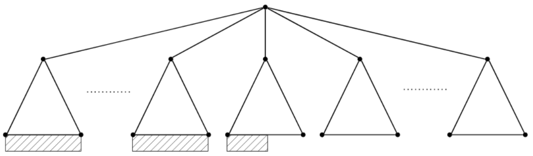

**Figure 2** A centroid tree after the reposition of subtrees. Lined rectangles represent leaves.

▶ **Theorem 6.** *Assuming $k$ is a constant, the difference in the total distance between an optimal $(k+1)$-degree tree $T$ and our centroid $(k+1)$-degree tree is $O(n^2 k \log k)$ while the total distance in the optimal $(k+1)$-degree tree is $\Omega(n^2 \log n)$.*

▶ **Remark 7.** *We can get a $k$-ary search tree out of $(k+1)$-degree centroid tree by rooting at some leaf and setting the identifiers correspondingly. We name such a tree a* centroid $k$-ary search tree*.*

*Since we consider the uniform workload we know that our centroid tree has the total cost of requests close to the optimal, i.e., misses by at most $O(n^2)$ while the total optimal cost is $\Omega(n^2 \log n)$. Thus, our centroid $k$-ary search tree has an approximation ratio $1 + O(\frac{1}{\log n})$.*

Then, it is quite straightforward to build that centroid tree.

▶ **Theorem 8.** *The centroid $k$-ary search tree can be built in $O(n)$ time.*

Finally, we show that the full $k$-ary tree also has total distance close to the cost of the optimal tree.

▶ **Lemma 9.** *The total distance in the full $k$-ary tree and the total distance in the centroid $(k+1)$-degree tree are both $n^2 \log_k n + O(n^2)$. That is, their total distance differs from the total distance in the optimal tree by $O(n^2)$.*

▶ **Remark 10.** *The results of the last Lemma show that the full and centroid trees are close to the optimal. However, in the uniform workload the centroid tree should have better total cost, since we split in the centroid vertex by $k+1$ balanced subtrees. In our experiments, we found that our* centroid *$k$-ary search tree* ***is indeed optimal*** *for all $n$ less than $10^3$ when $k$ is up to $10$, but we were not able prove its optimality formally.*

## 4 Online self-adjusting $k$-ary search tree networks

We present online algorithms for self-adjusting $k$-ary search tree networks. The first one is the $k$-ary SplayNet, which is a self-adjusting network based on a $k$-ary search tree and a generalization of SplayNet [22]. We prove that $k$-ary SplayNet has the same complexity bounds as SplayNet (we show its benefits experimentally in Section 5). The second one is $(k+1)$-SplayNet, a centroid-based structure, that is based on $k$-ary SplayNet and the centroid $k$-ary search tree network presented in the previous section.

### 4.1 $k$-ary SplayNet

In the literature, only one proposal for $k$-ary self-adjusting trees exists [23]. The tree rotations proposed in [23] cannot be directly generalized to SANs: multiple keys appear in each node and a node's keys change upon a rotation, so they cannot be used as network node identifiers and it is not clear how to maintain node identifiers across tree rotations. In this section, we propose new splay operations: $k$-semi-splay and $k$-splay. These rotations mimic the rotations in the binary splay tree and allow for persistent node identifiers, while re-shuffling routing arrays. For example, the routing array of the node with identifier $X$ in Figure 3 is $(a_1, a_2, a_3, \dots, a_{k-1})$ and key $X$ has value in between $a_3$ and $a_4$. Thus, any key can be located by the search property and the routing algorithm is the same as in SplayNet [22]: we rotate the source and destination to the lowest common ancestor and then route the request through a direct link. We now present splay operations that preserve the search property.

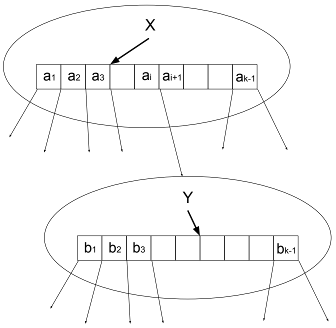

**Figure 3** The initial state for $k$-semi-splay.

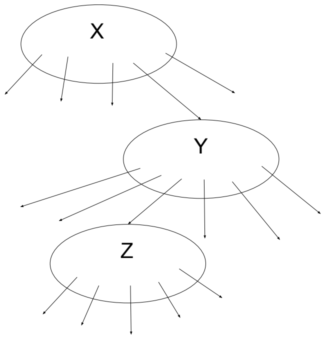

**Figure 4** Example state before $k$-splay.

We first present $k$-semi-splay, which generalizes the zig and zag operations in splay trees [24]. Suppose that we have two nodes: a node with id $X$ and a child node with id $Y$ (Figure 3). Our goal is to make $Y$ a parent of $X$. To that end, we merge the routing arrays from these nodes and search for the position for $X$ in this array. Then, we take $X$ as a key and some $k-1$ consecutive routing elements "covering" $X$ as a routing array, i.e., $X$ as the value lies

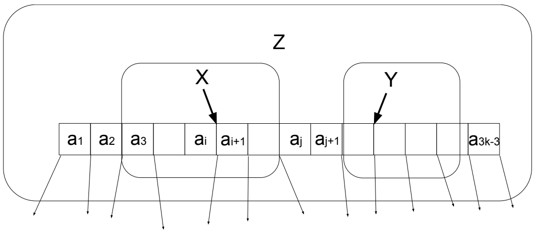

**Figure 5** $k$-splay rotation. The first case.

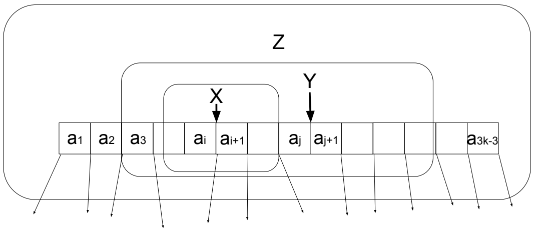

**Figure 6** $k$-splay rotation. The second case.

in the segment built on a consecutive set of $k-1$ elements. Finally, we set this node as a child of a new node with key $Y$ and the routing elements left.

Now, we explain the $k$-splay rotation, which generalizes the combination of two zig or zag operations in splay trees. Suppose we have three nodes with identifiers $X$, $Y$, and $Z$ (Figure 4), and we want to make $Z$ the top node. First, we merge the routing arrays of these three nodes into one array and find the positions of $X$ and $Y$ there. There are two cases: 1) $X$ and $Y$ are located distant to each other, i.e., separated by more than $k-1$ routing elements; or 2) $X$ and $Y$ are close to each other. In the first case, we make two new nodes: one with $X$ and $k-1$ consecutive routing elements "covering" $X$; and the one similar for $Y$. Finally, we set these new nodes as children of a node with key $Z$ and routing elements left (Figure 5). In the second case, we make two new nodes: the one with $Y$ is a parent of the new node with $X$. Then, we set the node with $Y$ as a child of a node with key $Z$ and routing elements left (Figure 6).

Using these rotations, we design a data structure called $k$-ary SplayNet which works similarly to binary SplayNet: upon serving a request between two keys, we use $k$-splay and $k$-semi-splay rotations to move the nodes to their lowest common ancestor, replacing it and one of its children. By that, after adjustments, the request can be served in constant time.

Here, we designed only two types of rotations. These rotations can be seen as the generalization of the zig/zag and zigzag rotations in binary splay tree. Now, we prove that a $k$-ary splay tree, i.e., all the routing requests are from the root, based on our rotations is statically-optimal. Please note that the lower bound to serve the search requests for $k$-ary splay tree is the same as for the standard splay tree [24]: suppose the lower bound is asymptotically better for $k$-ary splay tree, then we can represent $k$-ary tree as the splay tree with the constant multiplier $\log k$ on the total cost.

▶ Remark 11. The $k$-semi-splay and $k$-splay operations cannot be applied to routing-based trees. For example, after collapsing three nodes in $k$-splay (the second case, as in Figure 6) we might get routing keys $Z$, $Y$, and $X$ (in this order) as the leftmost ones in the joint array of routing keys. At that point, we cannot distribute the routing keys in the required order, since these keys $X$, $Y$, and $Z$ should be node identifiers and routing keys at the same time and $X$ should be on top. When taking $X$ as a node identifier and a routing key of the topmost node, we will get only $Y$ and $Z$ keys in the leftmost subtree, which restricts us from filling in $X$ with routing keys.

▶ **Theorem 12.** *The $k$-ary splay tree based on the $k$-semi-splay and $k$-splay rotations is statically-optimal. In other words, the total cost to serve $n_x$ search requests to node $x$ is $O(m + \sum_x n_x \cdot \log \frac{m}{n_x})$ where $m$ is the total number of requests.*

**Sketch.** The theorem can be proven in the almost identical way as the result for binary splay tree [24]. As in the splay tree proof, we present the potential of the node with key $v$ as $r(v) = \log w(v)$, where $w(v)$ is the total weight of all nodes in the subtree of $v$. Then, we can reprove the Access Lemma from [24], where the difference of the potential moving a node to the top does not exceed $3 \cdot (r(\texttt{root}) - r(v)) + 1$. This lemma holds since all the inequalities on the potentials of nodes from the proof in [24] remain the same since $k$-semi-splay changes the potential exactly as zig does; our first case rotation of $k$-splay changes the potential exactly as zig-zag does, and, finally, our second case rotation of $k$-splay changes the potential exactly as like zig-zig does. Thus, with the proved access lemma we can get the main theorem. ◀

Because of this theorem, the complexity bound proven for binary SplayNet in [22] holds for our $k$-ary SplayNet.

▶ **Theorem 13.** *$k$-ary SplayNet performs requests $\sigma = ((u_1, v_1), (u_2, v_2), \ldots, (u_m, v_m))$ with the cost of entropies of sources and destination: $O(\sum\limits_{x=1}^{n} a_x \cdot \log \frac{m}{a_x} + b_x \cdot \log \frac{m}{b_x})$, where $a_x$ is the number of requests with $x$ as a source and $b_x$ is the number of requests with $x$ as a destination.*

Moreover, since each node has $k$ children instead of two, we expect that the total routing costs in $k$-ary SplayNet are smaller than in the original SplayNet. The experimental comparison between them appears in Section 5.1.

We note that there can be more alternatives to $k$-semi-splay and $k$-splay. For example, we can take any $d$ connected nodes in the tree and modify them in a manner that the node with a chosen key will be in the topmost one after the update. This can be done as follows: 1) merge all $d$ routing arrays into one; 2) find the positions of our $d$ identifiers in this array; 3) choose some order of keys $k_1, k_2, \ldots, k_d$ in the nodes; 4) consider the $i$-th key $k_i$, take the $k-1$ consecutive routing keys “covering” $k_i$, and use them to form a new node with key $k_i$; 5) remove these routing elements from the total routing array and repeat the previous phase for next keys. At the end, the topmost node will contain the required key $k_d$. Thus, we can have different versions of $k$-ary SplayNet depending on the rotations we choose. In the remainder, we refer to $k$-ary SplayNet as any (black-box) implementation that maintains the search property and one identifier per node.

## 4.2 Application of the centroid heuristic

Our theoretical studies in Section 3.2 show that the total cost for the uniform workload of both structures, the full $k$-ary tree and the centroid one, is very close. But we know from our experiments that for $n$ up to $10^3$ and $k$ up to 10 the centroid tree is actually optimal. With this practical motivation, we designed an online heuristic based on the centroid idea. We present $(k+1)$-SplayNet, which is a centroid-based structure and the online self-adjusting equivalent of the static tree from Section 3.2. The topology is presented in Figure 8. We split the nodes in $k+1$ almost equal parts and specify two centroid nodes: $c_1$ and $c_2$. Centroid $c_2$ corresponds to the centroid from the previous section; its subtrees have $(n-2)/(k+1)$ nodes.

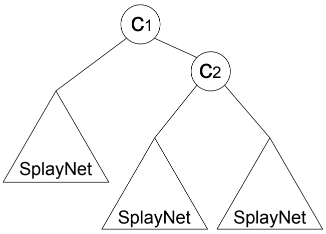

**Figure 7** 3-SplayNet structure

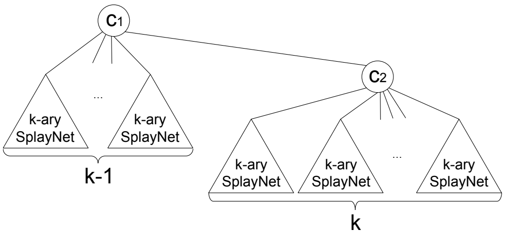

**Figure 8** $(k+1)$-SplayNet structure.

Centroid $c_1$ has $k-1$ children (in addition to $c_2$), that are $k$-ary SplayNets of size $[(n-2)/(k+1)]/(k-1)$. The $k$ children of $c_2$ are $k$-ary SplayNets of size $(n-2)/(k+1)$. When serving a request $(u, v)$, we $k$-splay $u$ and $v$ to their lowest common ancestor, as was done in $k$-ary SplayNet, but we never move nodes $c_1$ and $c_2$. That is, requests within the same subtree are served exactly as in $k$-ary SplayNet and for requests originating in different subtrees of $c_1$ and $c_2$, we splay the endpoints to their subtree roots and then we route the request via the path $u \to c_1 \to c_2 \to v$. The sets of nodes in the $2k-1$ subtrees remain intact, but these subtrees still can self-adjust. We study $(k+1)$-SplayNet and $k$-ary SplayNet experimentally in the next section.

## 5 Experimental evaluation

We have two presented approaches to evaluate: 1) we compare the cost of our $k$-ary SplayNet with the static balanced $k$-ary search tree, the optimal static routing-based $k$-ary search tree (Section 3.1), and the standard SplayNet, i.e., 2-ary SplayNet; 2) we compare our new 3-SplayNet based on the centroid heuristic ($(k+1)$-SplayNet for $k=2$) with the standard SplayNet and the two static data structures from item 1), for $k=2$. That is, in the first item we study the benefit of increasing $k$ across diverse workloads, while in the second item we compare the two heuristics of Section 4 and two static trees for $k=2$. In all our experiments, we set the routing and rotation costs to one.

The code for the algorithms was written in C++ and Python. We perform three types of experiments: (i) on the uniform workload with 100 nodes, (ii) on synthetic workloads with 1023 nodes and the temporal complexity parameter (the probability of repeating the last request [2]) taking the values 0.25, 0.5, 0.75, and 0.9, and (iii) on the data from three real-world datasets of datacenter network traces: a high performance computing (HPC) workload [11], a workload on ProjectToR [15], and a workload from Facebook's datacenter network traces [21]. We restrict all datasets to $10^6$ requests on: uniform workload with 100 nodes, HPC with 500 nodes, ProjectToR with 100 nodes, and Facebook with $10^4$ nodes.

### 5.1 $k$-ary SplayNet

In this section, we study how $k$-ary SplayNet performs on the chosen workloads. Our main goal is to investigate how the routing cost depends on $k$. For now, we assume that each of our rotations costs one. This assumption was also made in [13].

In Tables 1-8, we present how $k$-ary SplayNet works for $k \in [2, 10]$ against the static full and routing-based optimal $k$-ary trees. Note that we implicitly compare our data structures with the standard SplayNet which is the equivalent of our 2-ary SplayNet. In the first row, we show the total routing cost for 2-ary SplayNet and the relative cost of other $k$-ary SplayNets (the lower the better). In the second row, we show how our $k$-ary SplayNet works in comparison to the static full $k$-ary tree, i.e., the relative performance of our tree against the static one (the lower the number, the better our tree is). In the third row, we show how our structure works in comparison to the optimal static routing-based $k$-ary tree (the lower the better). We compare all workloads except for the uniform one, since we proved in Section 3.2 that full $k$-ary tree is almost optimal. The green color means that our tree performs better, otherwise, worse.

We make two main observations from the tables. Firstly, as expected, the higher the $k$ the lower the total routing cost in $k$-ary SplayNet. Secondly, with the low temporal locality, as for HPC, Facebook, 0.25, and 0.5 workloads, the full $k$-ary tree typically performs better on higher $k$, while the optimal routing-based $k$-ary tree works better but no more than 3 times.

The latter means that our data structure is constant-away from optimality in practice. This behaviour can be simply explained; the higher $k$ the better the cost for $k$-ary full trees on uniform workloads. On high locality, i.e., 0.75 and 0.9, our $k$-ary SplayNet outperforms both static trees. Also, we need to note that the algorithm building the optimal routing-based tree has high-complexity and we were not able to compute this tree for the Facebook workload.

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| SplayNet | 4798648 | 0.87x | 0.82x | 0.75x | 0.76x | 0.73x | 0.70x | 0.69x | 0.70x |
| Full Tree | 0.78x | 0.94x | 1.04x | 1.07x | 1.16x | 1.17x | 1.25x | 1.25x | 1.29x |
| Optimal Tree | 1.52x | 1.90x | 2.15x | 2.22x | 2.45x | 2.48x | 2.49x | 2.58x | 2.75x |

**Table 1** The comparison of $k$-ary SplayNet on HPC workload.

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| SplayNet | 3151626 | 0.93x | 0.91x | 0.87x | 0.84x | 0.86x | 0.86x | 0.84x | 0.83x |
| Full Tree | 0.40x | 0.49x | 0.46x | 0.52x | 0.70x | 0.50x | 0.58x | 0.57x | 0.92x |
| Optimal Tree | 1.45x | 1.81x | 2.09x | 2.10x | 2.08x | 2.20x | 2.22x | 2.22x | 2.25x |

**Table 2** The comparison of $k$-ary SplayNet on ProjectToR workload.

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| SplayNet | 12320225 | 0.85x | 0.77x | 0.74x | 0.72x | 0.70x | 0.70x | 0.68x | 0.67x |
| Full Tree | 0.69x | 0.87x | 0.94x | 1.00x | 1.07x | 1.11x | 1.15x | 1.19x | 1.28x |
| Optimal Tree | - | - | - | - | - | - | - | - | - |

**Table 3** The comparison of $k$-ary SplayNet on Facebook workload.

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| SplayNet | 1389359 | 0.82x | 0.75x | 0.71x | 0.69x | 0.68x | 0.68x | 0.65x | 0.62x |
| Full Tree | 0.99x | 1.15x | 1.23x | 1.30x | 1.37x | 1.39x | 1.47x | 1.51x | 1.55x |
| Optimal Tree | 1.75x | 2.12x | 2.32x | 2.49x | 2.64x | 2.71x | 2.88x | 2.99x | 3.03x |

**Table 4** The comparison of $k$-ary SplayNet on synthetic workload with temporal complexity parameter 0.25.

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| SplayNet | 963150 | 0.83x | 0.76x | 0.72x | 0.70x | 0.69x | 0.69x | 0.67x | 0.64x |
| Full Tree | 0.69x | 0.80x | 0.86x | 0.91x | 0.97x | 0.98x | 1.03x | 1.06x | 1.10x |
| Optimal Tree | 1.21x | 1.49x | 1.64x | 1.76x | 1.87x | 1.91x | 2.04x | 2.12x | 2.15x |

**Table 5** The comparison of $k$-ary SplayNet on synthetic workload with the temporal complexity parameter 0.5.

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| SplayNet | 530049 | 0.85x | 0.78x | 0.75x | 0.73x | 0.72x | 0.72x | 0.70x | 0.67x |
| Full Tree | 0.38x | 0.45x | 0.49x | 0.52x | 0.55x | 0.56x | 0.59x | 0.61x | 0.64x |
| Optimal Tree | 0.68x | 0.84x | 0.94x | 1.02x | 1.09x | 1.12x | 1.19x | 1.24x | 1.26x |

**Table 6** The comparison of $k$-ary SplayNet on synthetic workload with the temporal complexity parameter 0.75.

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| SplayNet | 271838 | 0.88x | 0.83x | 0.80x | 0.79x | 0.78x | 0.78x | 0.76x | 0.74x |
| Full Tree | 0.20x | 0.24x | 0.27x | 0.29x | 0.31x | 0.31x | 0.33x | 0.34x | 0.36x |
| Optimal Tree | 0.36x | 0.46x | 0.53x | 0.58x | 0.62x | 0.64x | 0.68x | 0.72x | 0.73x |

**Table 7** The comparison of $k$-ary SplayNet on synthetic workload with the temporal complexity parameter 0.9.

## 5.2 A case study of the centroid heuristic for $k = 2$

In this subsection, we study the online centroid heuristic experimentally for the case of $k = 2$. We compare our centroid-based 3-SplayNet (Figure 7) with the 2-ary SplayNet, which is the standard SplayNet. We implemented and compared them on different workloads that simulate real request patterns. As a result, it appears that on workloads with low temporal complexity 3-SplayNet works better than SplayNet.

We run the workloads described above on four different structures: 3-SplayNet, SplayNet, static full binary search tree, and static optimal binary search tree. In Table 8, one can see the average request cost by 3-SplayNet and the relative difference with other approaches. The green cell means that our 3-SplayNet is better and red means otherwise.

We observe that 3-SplayNet performs better or similarly to SplayNet on average and low temporal complexity workloads (0.25 and 0.5), while on high temporal complexity workloads (0.75 and 0.9) it works a bit worse. Also, 3-SplayNet outperforms SplayNet for the uniform, ProjecToR and Facebook workloads, but not for the HPC workload (higher locality than the other two real-world workloads). We interpret this as the effect of having fixed centroid nodes.

| | 3-SplayNet | SplayNet | Full Binary Net | Static Optimal Net |
|---|---|---|---|---|
| Uniform | 17.730 | x1.059 | x0.789 | x0.759 |
| HPC | 9.269 | x0.956 | x1.206 | x1.034 |
| ProjecToR | 2.865 | x1.132 | x3.040 | x0.800 |
| Facebook | 8.210 | x1.104 | x0.939 | x0.852 |
| Temporal 0.25 | 13.332 | x1.046 | x1.046 | x0.937 |
| Temporal 0.5 | 9.414 | x1.021 | x1.482 | x1.326 |
| Temporal 0.75 | 5.520 | x0.963 | x2.527 | x2.250 |
| Temporal 0.9 | 3.186 | x0.856 | x4.380 | x3.862 |

**Table 8** The comparison of 3-SplayNet with other known nets. Green means that 3-SplayNet is better.

# 6 Conclusion and future work

We presented online and offline algorithms for $k$-ary search tree networks. Specifically, we presented dynamic programming algorithms for computing an optimal static network for generic and uniformly distributed traffic. Then, we presented an online $k$-ary search tree network and a variant that utilizes our centroid structure. We proposed novel splay operations that are applicable in the context of self-adjusting networks. Our experimental results show: 1) the total routing cost for $k$-ary SplayNet is smaller than for standard SplayNet and is smaller than for full $k$-ary trees on traces with high locality; 2) on binary trees for real and synthetic traces of medium to low locality our centroid network outperforms SplayNet and its performance is always close to the best out of the algorithms tested. We believe that our work paves the way to new SANs for $k$-ary search tree networks for general and specific traffic patterns.

## References

1 Chen Avin, Marcin Bienkowski, Iosif Salem, Robert Sama, Stefan Schmid, and Paweł Schmidt. Deterministic self-adjusting tree networks using rotor walks. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 67–77. IEEE, 2022.

2 Chen Avin, Manya Ghobadi, Chen Griner, and Stefan Schmid. On the complexity of traffic traces and implications. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(1):1–29, 2020.

3 Chen Avin, Bernhard Haeupler, Zvi Lotker, Christian Scheideler, and Stefan Schmid. Locally self-adjusting tree networks. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 395–406. IEEE, 2013.

4 Chen Avin, Kaushik Mondal, and Stefan Schmid. Demand-aware network designs of bounded degree. *Distributed Computing*, pages 1–15, 2019.

5 Chen Avin, Kaushik Mondal, and Stefan Schmid. Demand-aware network design with minimal congestion and route lengths. *IEEE/ACM Transactions on Networking*, 30(4):1838–1848, 2022.

6 Chen Avin, Kaushik Mondal, and Stefan Schmid. Push-down trees: optimal self-adjusting complete trees. *IEEE/ACM Transactions on Networking*, 30(6):2419–2432, 2022.

7 Chen Avin, Iosif Salem, and Stefan Schmid. Working set theorems for routing in self-adjusting skip list networks. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 2175–2184. IEEE, 2020.

8 Chen Avin and Stefan Schmid. Toward demand-aware networking: A theory for self-adjusting networks. *ACM SIGCOMM Computer Communication Review*, 48(5):31–40, 2019.

9 Chen Avin and Stefan Schmid. Renets: Statically-optimal demand-aware networks. In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 25–39. SIAM, 2021.

10 Otavio Augusto de Oliveira Souza, Olga Goussevskaia, and Stefan Schmid. Cbnet: Minimizing adjustments in concurrent demand-aware tree networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 382–391. IEEE, 2021.

11 US DOE. Characterization of the doe mini-apps. `https://portal.nersc.gov/project/CAL/doe-miniapps.htm`, 2016.

12 Evgenii Feder, Anton Paramonov, Iosif Salem, Stefan Schmid, and Vitaly Aksenov. Toward self-adjusting k-ary search tree networks. *arXiv preprint arXiv:2302.13113*, 2023.

13 Evgeniy Feder, Ichha Rathod, Punit Shyamsukha, Robert Sama, Vitaly Aksenov, Iosif Salem, and Stefan Schmid. Toward self-adjusting networks for the matching model. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 429–431, 2021.

14 Evgeniy Feder, Ichha Rathod, Punit Shyamsukha, Robert Sama, Vitaly Aksenov, Iosif Salem, and Stefan Schmid. Lazy self-adjusting bounded-degree networks for the matching model. In *41th IEEE Conference on Computer Communications, INFOCOM 2020, Virtual Conference, May 2-5, 2022*. IEEE, 2022.

15 Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 216–229, 2016.

16 Chen Griner, Johannes Zerwas, Andreas Blenk, Manya Ghobadi, Stefan Schmid, and Chen Avin. Cerberus: The power of choices in datacenter topology design-a throughput perspective. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 5(3):1–33, 2021.

17 Matthew Nance Hall, Klaus-Tycho Foerster, Stefan Schmid, and Ramakrishnan Durairajan. A survey of reconfigurable optical networks. *Optical Switching and Networking*, 41:100621, 2021.

18 Charles Martel. Self-adjusting multi-way search trees. *Information Processing Letters*, 38(3):135–141, 1991.

**19** Bruna Peres, Otavio Augusto de Oliveira Souza, Olga Goussevskaya, Chen Avin, and Stefan Schmid. Distributed self-adjusting tree networks. *IEEE Transactions on Cloud Computing*, 2021.

**20** Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, et al. Jupiter evolving: Transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 66–85, 2022.

**21** Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.

**22** Stefan Schmid, Chen Avin, Christian Scheideler, Michael Borokhovich, Bernhard Haeupler, and Zvi Lotker. Splaynet: Towards locally self-adjusting networks. *IEEE/ACM Transactions on Networking*, 24(3):1421–1433, 2015.

**23** Murray Sherk. Self-adjusting k-ary search trees. *Journal of Algorithms*, 19(1):25–44, 1995.

**24** Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.